String Inference from Longest-Common-Prefix Array

Kärkkäinen, Juha

Schloss Dagstuhl - Leibniz-Zentrum für Informatik
2017

þÿ K ä r k k ä i n e n , J , P i t k o w s k i , M & P u g l i s i , S J 2 0 1 7 , S t r i n g I n f e r e n c e Longest-Common-Prefix Array . in I Chatzigiannakis , P Indyk , F Kuhn & A Muscholl (eds) , 44th International Colloquium on Automata, Languages, and Programming (ICALP 2017) . , 62 , Leibniz International Proceedings in Informatics (LIPIcs) , vol. 80 , Schloss Dagstuhl - Leibniz-Zentrum für Informatik , International Colloquium on Automata, Languages and Programming , Warsaw , Poland , 10/07/2017 . https://doi.org/10.4230/LIPIcs.ICALP.2017.62

http://hdl.handle.net/10138/232533
https://doi.org/10.4230/LIPIcs.ICALP.2017.62

String Inference from Longest-Common-Prefix Array*

Juha Kärkkäinen[1], Marcin Piątkowski[2], and Simon J. Puglisi[3]

1 **Helsinki Institute of Information Technology (HIIT), Helsinki, Finland; and Department of Computer Science, University of Helsinki, Helsinki, Finland**
juha.karkkainen@cs.helsinki.fi

2 **Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Toruń, Poland**
marcin.piatkowski@mat.umk.pl

3 **Helsinki Institute of Information Technology (HIIT), Helsinki, Finland; and Department of Computer Science, University of Helsinki, Finland**
simon.puglisi@cs.helsinki.fi

Abstract

The suffix array, perhaps the most important data structure in modern string processing, is often augmented with the longest common prefix (LCP) array which stores the lengths of the LCPs for lexicographically adjacent suffixes of a string. Together the two arrays are roughly equivalent to the suffix tree with the LCP array representing the tree shape.

In order to better understand the combinatorics of LCP arrays, we consider the problem of inferring a string from an LCP array, i.e., determining whether a given array of integers is a valid LCP array, and if it is, reconstructing some string or all strings with that LCP array. There are recent studies of inferring a string from a suffix tree shape but using significantly more information (in the form of suffix links) than is available in the LCP array.

We provide two main results. (1) We describe two algorithms for inferring strings from an LCP array when we allow a generalized form of LCP array defined for a multiset of cyclic strings: a linear time algorithm for binary alphabet and a general algorithm with polynomial time complexity for a constant alphabet size. (2) We prove that determining whether a given integer array is a valid LCP array is NP-complete when we require more restricted forms of LCP array defined for a single cyclic or non-cyclic string or a multiset of non-cyclic strings. The result holds whether or not the alphabet is restricted to be binary. In combination, the two results show that the generalized form of LCP array for a multiset of cyclic strings is fundamentally different from the other more restricted forms.

1998 ACM Subject Classification F.2.2 [Nonnumerical Algorithms and Problems] Pattern Matching, G.2.1 [Combinatorics] Combinatorial Algorithms, G.2.2 [Graph Theory] Eulerian cycles

Keywords and phrases LCP array, string inference, BWT, suffix array, suffix tree, NP-hardness

Digital Object Identifier 10.4230/LIPIcs.ICALP.2017.62

1 Introduction

For a string X of n symbols, the suffix array (SA) [23] contains pointers to the suffixes of X, sorted in lexicographical order. The suffix array is often augmented with a second array –

* The full version of this paper containing all the proofs and additional examples is available as [21], http://arxiv.org/abs/1606.04573.

44th International Colloquium on Automata, Languages, and Programming (ICALP 2017).
Editors: Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl;
Article No. 62; pp. 62:1–62:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the longest common prefix (LCP) array – storing the length of the longest common prefix between lexicographically adjacent suffixes; i.e., LCP[i] is the length of the LCP of suffixes $X[\mathrm{SA}[i]..n)$ and $X[\mathrm{SA}[i-1]..n)$. The two arrays are closely connected to the suffix tree [32] – the compacted trie of all the string's suffixes: the entries of SA correspond to the leaves of the suffix tree, and the LCP array entries tell the string depths of the lowest common ancestors of adjacent leaves, defining the shape of the tree. For decades these data structures have been central to string processing; see [4] for a history and an overview, and [1, 3, 15, 30, 26] for further details on myriad applications.

Given both the suffix and the LCP array, the corresponding string is unique up to renaming of the characters and is easy to reconstruct: zeros in the LCP array tell where the first character changes in the lexicographical list of the suffixes, and the suffix array tells how to permute those first characters to obtain the string. Given the suffix array without the LCP array, we can easily reconstruct a corresponding string where all characters are different, and it is not difficult to characterize the set of all strings with a given suffix array [5, 28, 22]. In essence, the suffix array determines a set of positions in the LCP array that must be zero. Specifically, for any i let j and k be integers such that $\mathrm{SA}[j] = \mathrm{SA}[i-1]+1$ and $\mathrm{SA}[k] = \mathrm{SA}[i]+1$. Then, if $k < j$, we must have $\mathrm{LCP}[i] = 0$. For any other position, we can freely and independently decide whether the value is zero or not, and as described above, the zero positions together with the suffix array determine the string.

In this paper, we consider the problem of similarly reconstructing strings from an LCP array without the suffix array. As mentioned above, the LCP array determines the shape of the suffix tree, i.e., the suffix tree without edge or leaf labels. String inference from the suffix tree shape has recently been considered by three different sets of authors [19, 6, 31]. However, all of them assume that the suffix tree is augmented with significant additional information, namely *suffix links*, which makes the task much easier. Indeed, our new algorithms essentially reconstruct suffix links from the LCP array. According to Cazaux and Rivals [6], the case without suffix links was considered but not solved in [27]. We are also aware that others have considered it but without success [2].

To fully define the problem, we have to specify what kind of strings we are trying to infer. Often suffix trees and suffix arrays are defined for *terminated strings* that are assumed to end with a special symbol \$ that is different from and lexicographically smaller than any other symbol. The alternative is an *open-ended string* where no assumption is made on the last symbol. For suffix and LCP arrays the only change from omitting the terminator symbol is dropping the first element (which is always zero in the LCP array), but the suffix tree can change considerably because some suffixes can be prefixes of other suffixes and thus are not represented by a leaf. Inferring open-ended strings from a suffix tree (with suffix links) is studied by Starikovskaya and Vildhøj [31], who show that any string can be appended by additional characters without changing the suffix tree shape (thus the term open-ended). However, such an extension can change the suffix and LCP arrays a great deal, i.e., with the arrays a string is never truly open-ended but has at least an implicit terminator.

To get rid of even an implicit terminator, we consider a third type of strings, *cyclic strings*, where we use rotations in place of suffixes. For a terminated string, replacing suffixes with rotations causes no changes to the suffix/rotation array or the LCP array. Thus any integer array that is a valid LCP array for a terminated string is always a valid LCP array for a cyclic string too, but the opposite is not true. For example, the LCP array for the cyclic string *aababa* is $(2, 1, 3, 0, 2)$, which is not a valid LCP array for any non-cyclic string. In this sense, the cyclic string case is strictly more general. An even more striking example is a non-primitive string, such as *abab*, that has two or more identical rotations. For reasons

explained below, instead of rotations we use *cyclic suffixes* which are infinite repetitions of rotations. Thus the LCP array for the cyclic string $abab$ is $(\omega, 0, \omega)$, where ω denotes the positions of two adjacent identical cyclic suffixes.

As a further generalization, we may have a joint suffix array for a collection of strings, where we have all suffixes of all strings in lexicographical order, and the corresponding LCP array. In the terminated version, each string is terminated with a distinct terminator symbol. If we have an LCP array for a collection of open-ended strings, adding the terminator symbols simply prepends one zero for each terminator. The LCP array for a collection of terminated strings is identical to the LCP array of the concatenation of the strings. Thus the generalization from single strings to string sets does not add to the set of valid LCP arrays for terminated strings, but it does for cyclic strings. For example the LCP array for a string set $\{aa, b\}$ is $(\omega, 0)$, which is not a valid LCP array for any single string. For multiple cyclic strings, it is important to use cyclic suffixes instead of rotations because the result can be different (e.g., the set $\{ab, aba\}$).

Now we are ready to formally define the problem of String Inference from LCP Array (SILA). In the decision version, we are given an array of integers (and possibly ω's) and asked if the array is a valid LCP array of some string. If the answer is yes, the reporting version may also output some such string, and possibly a characterization of all such strings. Different variants are identified by a prefix: S for a string set; T, O, or C for terminated, open-ended or cyclic; and B for a binary alphabet (where terminators are not counted). For example, BCSSILA stands for Binary Cyclic String Set Inference from LCP Array. As discussed above, and summarized in the following result (see [21] for the proof), the non-cyclic variants are essentially equivalent, but the cyclic variants are more general.

▶ **Proposition 1.** *There are polynomial time reductions from BTSILA to BOSILA, BTSSILA, BOSSILA, TSILA, OSILA, TSSILA, and OSSILA.*

Our Contribution

Our first result is a linear time algorithm for BCSSILA. For a valid LCP array the algorithm outputs a string, which is the Burrows-Wheeler transform (BWT) of the solution string set. This relies on a generalization of the BWT for multisets of cyclic strings developed in [24, 20]. There can be more than one multiset of strings with the same BWT but the class of such string collections is simple and well characterized in [20]. The algorithm also outputs a set of substring swaps such that applying any combination of the swaps on the BWT produces another BWT of a solution, and any BWT of a solution can be produced by such a combination of swaps. Thus we have a complete characterization of all solutions. The number of swaps can be linear and thus the number of distinct solutions can be exponential. We also present an algorithm for CSSILA, i.e., without a restriction on the alphabet size, that has a polynomial time complexity for any constant alphabet size.

Our second result is a proof, by a reduction from 3-SAT, that (the decision version of) BCSILA, and thus CSILA, is NP complete. Therefore, even though the BCSSILA algorithm produces a characterization of all solutions, it is NP hard to determine whether one of the solutions is a single string. Furthermore, we modify the reduction to prove that BTSILA is NP complete too. By Proposition 1, this shows that all variants of SILA mentioned above except (B)CSSILA are NP complete. Since CSSILA is in P for constant alphabet sizes, this leaves the complexity of CSSILA for larger alphabets as an open problem.

Related Work

String inference from partial information is a classic problem in string processing, dating back some 40 years to the work of Simon [29], where reconstructing a string from a set of its subsequences is considered. Since then, string inference from a variety of data structures has received a considerable amount of attention, with authors considering border arrays [12, 11, 10], parameterized border arrays [18], the Lyndon factorization [25], suffix arrays [5, 22], KMP failure tables [11, 13], prefix tables [7], cover arrays [9], and directed acyclic word graphs [5]. The motivation for studying most string inference problems is to gain a deeper understanding of the combinatorics of the data structures involved, in order to design more efficient algorithms for their construction and use.

A (somewhat tangentially) related result to ours is due to He et al. [16], who prove that it is NP hard to infer a string from the longest-previous-factor (LPF) array. It is well known that LPF is a permutation of LCP [8] but otherwise it is a quite different data structure. For example, it is in no way concerned with lexicographical ordering. Like our NP-hardness proof, He et al.'s reduction is from 3-SAT, but the details of each reduction appear to be very different. Moreover, their construction requires an unbounded alphabet while our construction works for a binary alphabet and thus for any alphabet.

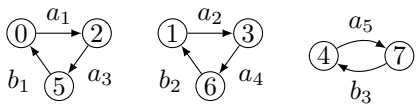
To the best of our knowledge, all of the previous string inference problems aim at obtaining a single non-cyclic string from some data structure, and we are the first to consider the generalizations to cyclic strings and to string sets, and as our results show, this makes a crucial difference. As explained in the next section, the generalizations arise naturally from the generalized BWT introduced in [24], which also played a central role in another recent result on the combinatorics of LCP arrays [20].

2 Basic notions

Let v be a string of length n and let $\widehat{v}$ be obtained from v by sorting its characters. The *standard permutation* [14, 17] of v is the mapping $\Psi_v : [0..n) \to [0..n)$ such that for every $i \in [0..n)$ it holds $\widehat{v}[i] = v[\Psi_v(i)]$ and for any $\widehat{v}[i] = \widehat{v}[j]$ the relation $i < j$ implies $\Psi_v(i) < \Psi_v(j)$. In other words, Ψ_v corresponds to the stable sorting of the characters. Let $C = \{c_i\}_{i=1}^{s}$ be the disjoint cycle decomposition of Ψ_v. We define the inverse Burrows–Wheeler transform IBWT as the mapping from v into a multiset of cyclic strings $W = \{\!\{w_i\}\!\}_{i=1}^{s}$ such that for any $i \in [1..s]$ and $j \in [0..|c_i|)$, $w_i[j] = v[\Psi_v(c_i[j])]$.

▶ **Example 2.** For $v = bbaabaaa$, we have $\mathrm{IBWT}(v) = \{\!\{aab, aab, ab\}\!\}$ as illustrated in the following table (showing $\widehat{v}$ and Ψ_v) and figure (showing the cycles of Ψ_v as a graph). The character subscripts are provided to make it easier to ensure stability.

i	0	1	2	3	4	5	6	7
$v[i]$	b_1	b_2	a_1	a_2	b_3	a_3	a_4	a_5
$\widehat{v}[i]$	a_1	a_2	a_3	a_4	a_5	b_1	b_2	b_3
$\Psi_v[i]$	2	3	5	6	7	0	1	4



The elements of W are primitive cyclic strings. *Cyclic* means that all rotations of a string are considered equal. For example, *aab*, *aba* and *baa* are all equal. A string is *primitive* if it is not a concatenation of multiple copies of the same string. For example, *aab* is primitive but *aabaab* is not. For any alphabet Σ, the mapping IBWT is a bijection between the set Σ^* of all (non-cyclic) strings and the multisets of primitive cyclic strings over Σ [24].

The set of positions of W is defined as the set of integer pairs $\mathrm{pos}(W) := \{\langle i,p\rangle \ : \ i \in [1..s],\ p \in [0..|w_i|)\}$. For a position $\langle i,p\rangle \in \mathrm{pos}(W)$ we define a *cyclic suffix* $W_{\langle i,p\rangle}$ as the infinite string that starts at $\langle i,p\rangle$, i.e., $W_{\langle i,p\rangle} = w_i[p]w_i[p+1 \bmod |w_i|]w_i[p+2 \bmod |w_i|],\ldots$. The multiset of all cyclic suffixes of W is defined as $\mathrm{suf}(W) := \{\!\{W_{\langle i,p\rangle} \ : \ \langle i,p\rangle \in \mathrm{pos}(W)\}\!\}$. We say that a string x occurs at position $\langle i,p\rangle$ in W if x is a prefix of the suffix $W_{\langle i,p\rangle}$.

The *(cyclic) suffix array* of a multiset of strings W is an array SA_W containing a permutation of $\mathrm{pos}(W)$ such that $W_{\mathrm{SA}_W[j-1]} \le W_{\mathrm{SA}_W[j]}$ for all $j \in [1..n)$. The *Burrows-Wheeler transform* (BWT) is a mapping from W into the string v defined as $v[j] = w_i[p-1 \bmod |w_i|]$, where $\langle i,p\rangle = \mathrm{SA}_W[j]$, i.e., $v[j]$ is the character preceding the beginning of the suffix $W_{\mathrm{SA}_W[j]}$. The BWT is the inverse of IBWT [24, 20].

The *longest-common-prefix array* $\mathrm{LCP}_W[1..n)$ is defined as $\mathrm{LCP}_W[j] = \mathrm{lcp}\big(W_{\mathrm{SA}_W[j-1]}, W_{\mathrm{SA}_W[j]}\big)$ for $0 < j < n$, where $\mathrm{lcp}(x,y)$ is the length of the longest common prefix between the strings x and y.

▶ **Example 3.** For $W = \{\!\{ab,\ aab,\ aab\}\!\}$ we have

$$\begin{aligned}\mathrm{suf}(W) &= \{\!\{(aab)^\omega, (aab)^\omega, (aba)^\omega, (aba)^\omega, (ab)^\omega, (baa)^\omega, (baa)^\omega, (ba)^\omega\}\!\}\\ \mathrm{SA}_W &= \big[\langle 2,0\rangle,\ \langle 3,0\rangle,\ \langle 2,1\rangle,\ \langle 3,1\rangle,\ \langle 1,0\rangle,\ \langle 2,2\rangle,\ \langle 3,2\rangle,\ \langle 1,1\rangle\big]\\ \mathrm{LCP}_W &= \big[\omega, 1, \omega, 3, 0, \omega, 2\big].\end{aligned}$$

The suffixes represented by the suffix array entries can also be expressed as follows.

▶ **Lemma 4.** *For $i \in [0..n)$, $W_{\mathrm{SA}_W[i]} = \widehat{v}[i]\widehat{v}[\Psi_v(i)]\widehat{v}[\Psi_v^2(i)]\widehat{v}[\Psi_v^3(i)]\ldots$.*

2.1 Intervals

Many algorithms on suffix arrays and LCP arrays are based on iterating over a specific types of array intervals. Next, we define these intervals and establish their key properties. For proofs and further details, we refer to [1, 26].

Let $v \in \{a,b\}^n$ and $W = IBWT(v)$. Let $\mathrm{SA} = \mathrm{SA}_W$ be the suffix array and $\mathrm{LCP} = \mathrm{LCP}_W$ the LCP array of W. Note that from now on, we will assume a binary alphabet.

▶ **Definition 5** (x-interval)**.** An interval $[i..j)$, $0 \le i \le j \le n$, is called the x-interval ($x \in \Sigma^*$) if and only if (1) x is not a prefix of $W_{\mathrm{SA}[i-1]}$ (or $i = 0$), (2) x is a prefix of $W_{\mathrm{SA}[k]}$ for all $k \in [i..j)$, and (3) x is not a prefix of $W_{\mathrm{SA}[j]}$ (or $j = n$).

In other words, in the suffix array the x-interval $\mathrm{SA}[i..j)$ consists of all suffixes of W with x as a prefix. Thus the size $j-i$ of the interval is the number of occurrences of x in W, which we will denote by n_x.

▶ **Definition 6** (ℓ-interval)**.** An interval $[i..j)$, $0 \le i < j \le n$, is called an ℓ-interval ($\ell \in \mathbb{N} \cup \{\omega\}$) if and only if (1) $LCP[i] < \ell$ (or $i = 0$), (2) $\min \mathrm{LCP}[i+1..j) = \ell$ (where $\min \mathrm{LCP}[j..j) = \omega$), and (3) $LCP[j] < \ell$ (or $j = n$).

▶ **Lemma 7.** *Every nonempty x-interval is an ℓ-interval for some (unique) $\ell \ge |x|$. Every ℓ-interval is an x-interval for some string x of length ℓ.*

▶ **Corollary 8.** *If an x-interval $[i..j)$ is an ℓ-interval for $\ell > |x|$, there exists a (unique) string y of length $\ell - |x|$ such that $[i..j)$ is the xy-interval.*

Thus the ℓ-intervals represent the set of all distinct x-intervals. This and the fact that the total number of ℓ-intervals is $\mathcal{O}(n)$ are the basis of many efficient algorithms for suffix arrays, see e.g., [1, 26].

Algorithm 1: Infer BWT from an LCP array.

Input: an array $\mathrm{LCP}[1..n)$ of integers and ω's
Output: a string $v \in \{a,b\}^n$ such that $\mathrm{LCP}_{\mathrm{IBWT}(v)} = \mathrm{LCP}$ together with a set S of swap intervals, or **false** if there is no such string v

```
1  S := ∅;
2  preprocess LCP for RMQs;
3  k := RMQ_LCP[1..n);
4  if LCP[k] ≠ 0 then
5      if LCP[k] = ω then return a^n, ∅;
6      else return false;
7  InferInterval([0,n), [0,k), [k,n));
8  compute W = IBWT(v), SA_W, and LCP_W;
9  if LCP_W ≠ LCP then return false;
10 return v, S;
```

3 Algorithm for BCSSILA

We are now ready to describe the algorithm for string inference from an LCP array. Given an LCP array $\mathrm{LCP}[1..n)$, our goal is to construct a string $v \in \{a,b\}^n$ such that $\mathrm{LCP} = \mathrm{LCP}_{\mathrm{IBWT}(v)}$. At first, we assume that such a string v exists, and consider later what happens if the input is not a valid LCP array.

Let $\mathrm{RMQ}_{\mathrm{LCP}}[i..j)$ denote the *range minimum query* over the LCP array that returns the position of the minimum element in $\mathrm{LCP}[i..j)$, i.e., $\mathrm{RMQ}_{\mathrm{LCP}}[i..j) = \arg\min_{k\in[i..j)} \mathrm{LCP}[k]$. The LCP array is preprocessed in linear time so that any RMQ can be answered in constant time (see for instance [26]). Then any x-interval can be split into two subintervals as shown in the following result.

▶ **Lemma 9.** *Let $[i..j)$ be an x-interval and an ℓ-interval for $\ell < \omega$, and let $k = \mathrm{RMQ}_{\mathrm{LCP}}[i+1..j)$. Then, for some string y of length $\ell - |x|$, $[i..k)$ is the xya-interval and $[k..j)$ is the xyb-interval.*

This approach makes it easy to recursively enumerate all ℓ-intervals. We will also keep track of ax- and bx-intervals together with any x-interval, even if we do not know x precisely. From the intervals we can determine the numbers of occurrences, n_{ax} and n_{bx}, which are useful in the inference of v:

▶ **Lemma 10.** *Let $[i..j)$ be the x-interval. Then $v[i..j)$ contains exactly n_{ax} a's and n_{bx} b's.*

In particular, when either n_{ax} or n_{bx} drops to zero, we have fully determined $v[i..j)$ for the x-interval $[i..j)$. In such a case, the LCP array intervals have to satisfy the following property.

▶ **Lemma 11.** *Let $[i_y..j_y)$ be the y-interval for $y \in \{x, ax, bx\}$. If $n_{ax} = j_{ax} - i_{ax} = 0$, then $\mathrm{LCP}[i_{bx}+1..j_{bx}) = 1 + \mathrm{LCP}[i_x+1..j_x)$, where $1+A$, for an array A, denotes adding one to all elements of A. Symmetrically, if $n_{bx} = 0$, then $\mathrm{LCP}[i_{ax}+1..j_{ax}) = 1 + \mathrm{LCP}[i_x+1..j_x)$.*

The main procedure is given in Algorithm 1. The main work is done in the recursive procedure InferInterval given in Algorithm 2. The procedure gets as input the x-, ax- and bx-intervals for some (unknown) string x, splits the x-interval into xya- and xyb-subintervals

Algorithm 2: InferInterval($[i_x..j_x)$, $[i_{ax}..j_{ax})$, $[i_{bx}..j_{bx})$).

Input: (nonempty) x-, ax- and bx-intervals
Output: Set $v[i_x..j_x)$ and add the swap intervals within $[i_x..j_x)$ to S

```
1  k_x := RMQ_LCP[i_x + 1..j_x); m_x := LCP[k_x];
2  if j_ax − i_ax = 1 then k_ax := i_ax; m_ax := ω;
3  else k_ax := RMQ_LCP[i_ax + 1..j_ax); m_ax := LCP[k_ax];
4  if j_bx − i_bx = 1 then k_bx := i_bx; m_bx := ω;
5  else k_bx := RMQ_LCP[i_bx + 1..j_bx); m_bx := LCP[k_bx];
6  if m_ax > m_x + 1 and m_bx > m_x + 1 then
7      if LCP[i_ax + 1..j_ax) = 1 + LCP[i_x + 1..k_x) then
8          v[i_x..k_x) = aa...a; v[k_x..j_x) = bb...b;
9          if LCP[i_ax + 1..j_ax) = 1 + LCP[k_x + 1..j_x) then add [i_x..j_x) to S ;
10     else
11         v[i_x..k_x) = bb...b; v[k_x..j_x) = aa...a;
12 else if m_ax > m_x + 1 then
13     if k_bx − i_bx = k_x − i_x then
14         v[i_x..k_x) = bb...b;
15         InferInterval([k_x..j_x), [i_ax..j_ax), [k_bx..j_bx));
16     else
17         v[k_x..j_x) = bb...b;
18         InferInterval([i_x..k_x), [i_ax..j_ax), [i_bx..k_bx));
19 else if m_bx > m_x + 1 then
20     if k_ax − i_ax = k_x − i_x then
21         v[i_x..k_x) = aa...a;
22         InferInterval([k_x..j_x), [k_ax..j_ax), [i_bx..j_bx));
23     else
24         v[k_x..j_x) = aa...a;
25         InferInterval([i_x..k_x), [i_ax..k_ax), [i_bx..j_bx));
26 else
27     InferInterval([i_x..k_x),[i_ax..k_ax),[i_bx..k_bx));
28     InferInterval([k_x..j_x),[k_ax..j_ax),[k_bx..j_bx));
```

based on Lemma 9, and tries to split ax- and bx-intervals similarly. If all subintervals are nonempty, the algorithm processes the two subinterval triples recursively (lines 27 and 28).

When trying to split the ax-interval, the result may be, for example, that the $axya$-interval is empty. In this case, we do not need to recurse on the xya-interval since the corresponding part of v must be all b's. The algorithm recognizes the emptiness of $axya$- or $axyb$-interval by the fact that $m_{ax} > m_x + 1$, but the problem is to decide which is the empty one. In most cases, this can be determined by comparing the sizes of the different subintervals or even the actual LCP-intervals (see Lemma 11).

There is one case, where the algorithm is unable to determine the empty subintervals, which is when $\mathrm{LCP}[i_{ax}{+}1..j_{ax}) = \mathrm{LCP}[i_{bx}{+}1..j_{bx}) = 1{+}\mathrm{LCP}[i_x{+}1..k_x) = 1{+}\mathrm{LCP}[k_x{+}1..j_x)$. Then, either the $axya$- and $bxyb$-intervals are empty or the $axyb$- and $bxya$-intervals are empty, but there is no way of deciding between the two cases. It turns out that both are valid

choices. The algorithm sets v according to one choice (line 8) but records the alternative choice by adding the interval to the set S. In such a case, the string xy is called a *swap core* and the xy-interval (equal to the x-interval) is called a *swap interval*.

For each swap interval $[i..j)$, the algorithm sets $v[i..k) = aa\dots a$ and $v[k..j) = bb\dots b$, where $k = (i+j)/2$, but swapping the two halves would be an equally good choice. Therefore, if the output of the algorithm contains s swap intervals, it represents a set of 2^s distinct strings. The following lemma shows that the swaps indeed do not affect the LCP array (the proof can be found in [21]).

▶ **Lemma 12.** *Let $v \in \{a,b\}^n$, $W = \mathrm{IBWT}(v)$, $\mathrm{SA} = \mathrm{SA}_W$ and $\mathrm{LCP} = \mathrm{LCP}_W$. Let x be a string that occurs in W and satisfies: (1) $\mathrm{LCP}[i_{xa}+1..j_{xa}) = \mathrm{LCP}[i_{xb}+1..j_{xb})$, and (2) $v[i_{xa}..j_{xa}) = aa\dots a$ and $v[i_{xb}..j_{xb}) = bb\dots b$, where $[i_z..j_z)$ is the z-interval for $z \in \{xa, xb\}$. Let v' be the same as v except that $v'[i_{xa}..j_{xa}) = bb\dots b$ and $v'[i_{xb}..j_{xb}) = aa\dots a$. Then $\mathrm{LCP}_{\mathrm{IBWT}(v')} = \mathrm{LCP}$.*

▶ **Theorem 13.** *Algorithm 1 computes in linear time a representation of the set of all strings $v \in \{a,b\}^*$ such that $\mathrm{LCP}_{\mathrm{IBWT}(v)}$ is the input array, or returns false if no such string exists.*

Proof. Since the algorithm verifies its result (lines 9 and 10), it will return false if the input is not a valid LCP array. Given a valid LCP array, Algorithm 2 sets all elements of v since it recurses on any subinterval that it doesn't set. All the choices made by the algorithm are forced by the lemmas in this and the previous section. The swap intervals record all alternatives in the cases where the content of v could not be fully determined, and all of those alternatives have the same LCP array by Lemma 12. It is also easy to see that the algorithm runs in linear time. ◀

4 Coupling Constrained Eulerian Cycle

We will now set out to prove the NP-completeness of the single string inference problems BCSILA and BTSILA. The proofs are done by a reduction from 3-SAT via an intermediate problem called Coupling Constrained Eulerian Cycle (CCEC) described in this section.

Consider a directed graph G of degree two, i.e., every vertex in G has exactly two incoming and two outgoing edges. If G is connected, it is Eulerian. An Eulerian cycle can pass through each vertex in two possible ways, which we call the *straight state* and the *crossing state* of the vertex as illustrated here:

We consider each vertex to be a *switch* that can be flipped between these two states. The combination of vertex states is called the *graph state*. For a given graph state, the paths in the graph form, in general, a collection of cycles. The Eulerian cycle problem can then be stated as finding a graph state such that there is only a single cycle; we call such a graph state Eulerian.

In the *Coupling Constrained Eulerian Cycle (CCEC) problem*, we are given a graph as described above, an initial graph state, and a partitioning of the set of vertices. If we flip a vertex state, we must simultaneously flip the states of all the vertices in the same partition, i.e., the vertices in a partition are coupled. A graph state that is achievable from the initial state by a set of such *partition flips* is called a *feasible state*. The CCEC problem is to determine if there exists a feasible graph state that is Eulerian.

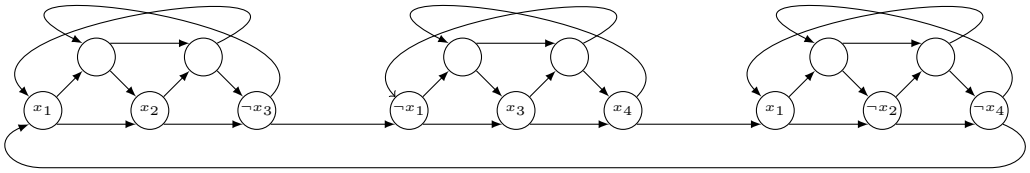


Figure 1 The CCEC graph corresponding to a 3-CNF formula $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_4)$.

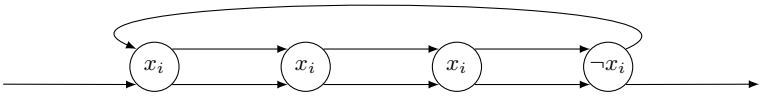
▶ **Theorem 14.** *CCEC is NP-complete.*

Proof. The proof is by reduction from 3-SAT. To obtain a CCEC graph from a 3-CNF formula, a gadget of five vertices is constructed from each clause and these gadgets are connected by a cycle. In each gadget, three of the vertices are labeled by the literals of the corresponding clause; the other two are called free vertices. See Fig. 1 for an illustration.

Each labeled vertex is in a straight state if the labeling literal is false and in a crossing state if the literal is true; their initial state corresponds to some arbitrary truth assignment to the variables. For each variable x_i, there is a vertex partition consisting of all vertices labeled by x_i or $\neg x_i$, so that flipping this partition corresponds to changing the truth value of x_i. Each free vertex forms a singleton partition and has an arbitrary initial state. Thus a graph state is feasible iff the labeled vertex states correspond to some truth assignment.

If a clause is false for a given truth assignment, the labeled vertices in the corresponding gadget are all in a straight state. This separates a part of the gadget from the main cycle and thus the graph state is not Eulerian. If a clause is true, at least one of the labeled vertices in the gadget is in a crossing state. Then we can always choose the state of the free vertices so that the full gadget is connected to the main cycle. Thus there exists a feasible Eulerian graph state iff there exists a truth assignment to the variables that satisfies all clauses. ◀

For purposes that will become clear later, we modify the above construction by adding some extra components to the graph without changing the validity of the reduction. Specifically, for each variable x_i in the 3-CNF formula we add the following gadget to the main cycle:



The vertices in the gadget are treated similarly to the other vertices in the graph: they belong to the partition with the other vertices labeled by x_i or $\neg x_i$, and the initial state is determined by the truth value of the labeling literal. It is easy to see that the gadget will be fully connected to the main cycle whether x_i is true or false. Thus the extra gadgets have no effect on the existence of an Eulerian cycle. Finally, we insert to the main cycle a single vertex labelled y with a self loop and forming a singleton partition.

5 BCSILA to CCEC

The next step is to establish a connection between the BCSILA and CCEC problems by showing a reduction from BCSILA to CCEC. Although the direction of the reduction is opposite to what we want, this construction plays a key role in the analysis of the main construction described in the next section.

Given a BCSILA instance (an integer array), we use Algorithm 1 to produce a representation of a set V of strings. The problem is then to decide if there exists $v \in V$ such that

IBWT(v) is a single (cyclic) string. We will write V as a string with brackets marking the swaps. For example, $V = b[ab][ab]a = \{bababa, babbaa, bbaaba, bbabaa\}$. In Example 2, we saw that the inverse BWT of a string $v \in V$ can be represented as a graph G_v where the vertices are labeled by positions in v and there is an edge between vertices i and j if, for some character $c \in \{a, b\}$ and some integer k, $\widehat{v}[i] = c$ is the kth occurrence of c in $\widehat{v}$ and $v[j] = c$ is the kth occurrence of c in v. Such an edge (i, j) is labeled by c_k. Note that $\forall v \in V$, $\widehat{v}$ is the same; we will denote it by $\widehat{V}$. We form a generalized graph G_V as a union of the graphs G_v, $v \in V$.

Consider a_k (the kth a) in $\widehat{V}$, say at position i. If a_k is outside any swap region in V, say at position j, there is a single edge (i, j) in G_V labeled by a_k. If a_k is within a swap region in V, it has two possible positions in the strings $v \in V$, say j and j'. That same pair of positions are also the possible positions of some b, say $b_{k'} = \widehat{V}[i']$. Then g_v has two edges, (i, j) and (i, j'), labeled with a_k and two edges, (i', j) and (i', j'), labeled with $b_{k'}$. The positions/vertices j and j' are called a *swap pair*.

To obtain a CCEC graph $\widetilde{G}_V$, we make two modifications to G_V. First, we merge each swap pair into a single vertex. Each merged vertex now has two incoming and two outgoing edges and all other vertices have one incoming and one outgoing edge. Second, we remove all vertices with degree one by concatenating their incoming and outgoing edges.

The initial state of the vertices in $\widetilde{G}_V$ is set so that the cycles in $\widetilde{G}_V$ correspond to the cycles in G_v for some $v \in V$. Two vertices in $\widetilde{G}_V$ belong to the same partition if their labels belong to the same swap interval in V. Then we have a one-to-one correspondence between swaps in V and partition flips in $\widetilde{G}_V$. If this CCEC instance has a solution, the Eulerian cycle spells a single string realizing the input LCP array. If the CCEC instance has no solution, the original BCSILA problem has no solution either.

6 BCSILA is NP-Complete

We are now ready to show that BCSILA is NP-complete using the reduction chain 3-SAT $\rightarrow$ CCEC $\rightarrow$ BCSILA. The first step was described in Section 4, and we will next describe the second. The latter reduction is not a general reduction from an arbitrary CCEC instance but works only for a CCEC instance obtained by the first reduction (including the extra gadgets).

The above BCSILA to CCEC reduction transforms each pair of swapped positions into a vertex and each swap interval into a vertex partition. Our construction creates a BCSILA instance such that the resulting BWT has the necessary swaps to produce the CCEC instance vertices and partitions. However, the BWT also has some unwanted swaps producing spurious vertices, but we will show that these spurious vertices do not invalidate the reduction.

Starting from a CCEC instance, we construct a set of cyclic strings and obtain the BCSILA instance as the LCP array of that string set. The construction associates two strings to each vertex and the cyclic strings are formed by concatenating the vertex strings according to the cycles in the graph in its initial state. The two passes of the cycles through a vertex must use different strings but it does not matter which pass uses which string.

Let n be the number of vertices in the CCEC graph and let m be the number of vertex partitions. We number the vertices from 1 to n and the partitions from 1 to m. The biggest partition number is assigned to the partition with the vertex y, the second biggest to the partition corresponding to the variable x_1, the third biggest to variable x_2, and so on. The three biggest vertex numbers are assigned to the vertices labeled x_1 in the extra gadget for the variable x_1, the next three biggest to the extra gadget vertices labeled x_2 and so on.

Within each extra gadget, the biggest number is assigned to the middle one of the three vertices. The strings associated with a vertex are ba^kba^{m+2h} and bba^kbba^{m+2h-1}, where k is the partition number and h is the vertex number. This completes the description of the transformation from a CECC instance to a BCSILA instance.

Let us now analyze the transformation by changing the BCSILA instance back to a CCEC instance using the construction of the preceding section. Specifically, we will analyze the swaps in the BWT produced from the LCP array. Let W be the set of cyclic strings constructed from the CCEC instance, and let V be the BWT with swaps constructed from LCP_W. An interval $[i..j)$ in V is a swap interval if and only if (1) $[i..j)$ is an x-interval for a string x such that either $occ(axa) = occ(bxb) = occ(x)/2$ or $occ(axb) = occ(bxa) = occ(x)/2$, where $occ(y)$ is the number of occurrences of y in W, and (2) $\mathrm{LCP}_W[i+1..k) = \mathrm{LCP}_W[k+1..j)$, where $k = (i+j)/2$. If $[i..j)$ is a swap interval, the string x is called its *swap core*. Our goal is to identify all swap cores.

Let us first consider strings of the form $x = ba^kb$. If $k > m$, $occ(x) \leq 1$ and x cannot be a swap core. For $k \in [1..m]$, x is always a swap core and corresponds to the CCEC partition numbered k. Let $v = \mathrm{BWT}(W)$ and let V' be v together with the swaps for cores of the form $x = ba^kb$, $k \in [1..m]$. It is easy to verify that a CCEC instance constructed from V' as described in the previous section is identical to the original CCEC instance. Thus, if there were no other swap cores, we would have a perfect reduction.

Unfortunately, there are other swap cores. A systematic examination of all strings (see [21] for details) shows that the other swap cores must be of the following forms: ba^{m+2n-1}, $a^{m+2n-1}b$, a^mba^m, a^mbba^m,a^kba^h, a^kbba^h, $a^kba^iba^h$ and $a^kbba^ibba^h$. Furthermore, it shows that each such swap core has exactly two occurrences, which means that the values k and/or h have to be sufficiently large. Each extra swap core adds a free vertex that is connected to the graph by making two existing edges to pass through the new vertex. Because of the way we chose to assign the biggest partition and vertex numbers, all the additional connections are within the extra gadgets, which does not change the existence of an Eulerian cycle. This completes the proof.

▶ **Theorem 15.** *BCSILA is NP-complete.*

7 BTSILA is NP-Complete

We will now show that BTSILA is NP-complete by modifying the above reduction for BCSILA to include a single terminator symbol \$ in the strings. The modification is applied to the set W of cyclic strings derived from the CCEC instance such that LCP_W is the BCSILA instance. Specifically, we replace the (unique) occurrence of a^{m+2n}, which is the longest consecutive run of a's, with $a^{m+2n+1}\$a^{m+2n}$ to obtain $W_\$$ and $\mathrm{LCP}_{W_\$}$. We will show that $\mathrm{LCP}_{W_\$}$ is a yes-instance of CSILA iff LCP_W is a yes-instance of BCSILA. Furthermore, if a cyclic string u is a solution to the CSILA instance, i.e., $\mathrm{LCP}_u = \mathrm{LCP}_{W_\$}$, then $\mathrm{LCP}_v = \mathrm{LCP}_{W_\$}$, where v is the rotation of u ending with \$ interpreted as a terminated string. Thus $\mathrm{LCP}_{W_\$}$ is a yes-instance of BTSILA iff it is a yes-instance of CSILA iff LCP_W is a yes-instance of BCSILA.

In general, adding even a single occurrence of a third symbol complicates the inference of the BWT from the LCP array and means that the set of equivalent BWTs can no more be described by a set of swaps. Consider how the operation of the procedure InferInterval (Algorithm 2) changes. First, it gets an extra $\$x$-interval as an input in addition to x-, ax- and bx-intervals. Second, the x-interval may be split into three subintervals, $xy\$$-, xya- and xyb-intervals, instead of two (which happens when the LCP interval contains two identical

minima). This leads to many more combinations to consider, and some of those combinations are more complicated.

Fortunately, in our case, having the single \$ surrounded by the two longest runs of a's simplifies things, and we will describe a modification of InferInterval to handle this case. Every call to InferInterval belongs to one of the following three types: (1) the x-interval is split into two and the $\$x$-interval is empty, (2) the x-interval is split into two and the $\$x$-interval is non-empty, and (3) the x-interval is split into three. The first case needs no modification at all. The other two cases mean that either $\$x$ or $x\$$ occurs in the produced string set, and since this property is not affected by swaps (or the threeway permutations described below), one of them occurs in every produced string set including $W_\$$. Since x must occur at least twice, one of the latter two cases happens iff $x = a^k$ for some $k \in [0..m + 2n]$. Although in general InferInterval cannot always know x, it is easy to keep track of x when $x = a^k$.

When InferInterval is called with $x = a^k$ for $k \leq m + 2n - 2$, the x-interval and the ax-interval are always split into three, the bx-interval is split into two, and there is a $\$x$-interval of size one. In general, we might not know whether the two subintervals of bx-interval are $bx\$$- and bxa-, or $bx\$$- and bxb-, or bxa- and bxb-intervals. However, since $x\$$- and $ax\$$-intervals both have size one, there can be no $bx\$$-interval, and thus all the subintervals can be uniquely determined and recursed on. When $x = a^{m+2n-1}$, the x-interval has size five and is split into three with the middle part (xa-interval) having size three. The ax interval has size three and is split into three. In this case too, only one combination of subintervals is possible.

When $x = a^{m+2n}$, the x-interval has size three and is split into three, and the $\$x$-, ax- and bx-intervals have size one. Therefore, the x-interval in the BWT contains some permutation of the three characters and all permutations are valid. This threeway permutation adds to the variation provided by the swaps in other parts of the BWT. A more careful analysis shows that the BWT x-interval of

- $\$ab$ or $\$ba$ implies an occurrence of $\$x\$$ which is only possible if $x\$$ is a separate string;
- $ba\$$ implies an occurrence of axa which is only possible if a single a is separate string;
- $a\$b$ implies occurrences of $ax\$$ and $\$xa$ which is only possible if $ax\$$ is a separate string;
- $ab\$$ implies an occurrence of $ax\$xb$; and
- $b\$a$ implies an occurrence of $bx\$xa$.

A single string solution is only possible in the last two cases, and any such solution corresponds to a solution for the BCSILA instance LCP_W (obtained by replacing $ax\$x$ or $x\$ax$ with x). Hence $\mathrm{LCP}_{W_\$}$ is a yes-instance of CSILA, and thus of BTSILA, if and only if LCP_W is a yes-instance of BCSILA, which proves the following result.

▶ **Theorem 16.** *BTSILA is NP-complete.*

8 Algorithm for CSSILA

In all of the above, we have assumed a binary alphabet (excluding the single symbol \$). In this section, we consider the CSSILA problem (i.e. Cyclic String Set Inference from LCP Array) without a restriction on the alphabet size (see [21] for more details).

Let $\mathcal{L}[1..n)$ be an instance of the CSSILA problem, i.e., an array of integers (and possibly ω's). Let $\sigma - 1$ be the number of zeroes in $\mathcal{L}$, and Σ an alphabet of size σ. As with the binary BCSSILA problem, we describe an algorithm that outputs a representation of the set $W_\mathcal{L} = \{w \in \Sigma^n : \mathrm{LCP}_{\mathrm{IBWT}(w)} = \mathcal{L}\}$; in this case the representation is an automaton that accepts $W_\mathcal{L}$. We show the following result.

▶ **Theorem 17.** *Given an array $\mathcal{L}[1..n)$ of integers (and possibly ω's) containing $\sigma-1$ zeroes, we can construct a deterministic finite automaton recognizing $W_{\mathcal{L}}$ in time $O(\sigma^2 2^\sigma(\frac{n}{\sigma}+1)^\sigma)$ and space $O(\sigma 2^\sigma(\frac{n}{\sigma}+1)^\sigma)$.*

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.

2 Amihood Amir. Personal communication, String Masters in Rouen, France, 3–5 February, 2014.

3 Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO Advanced Science Institutes Series F12, pages 85–96. Springer-Verlag, 1985. doi:10.1007/978-3-642-82456-2_6.

4 Alberto Apostolico, Maxime Crochemore, Martin Farach-Colton, Zvi Galil, and S. Muthukrishnan. 40 years of suffix trees. *Communications of the ACM*, 59(4):66–73, 2016. doi:10.1145/2810036.

5 Hideo Bannai, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Inferring strings from graphs and arrays. In *Proceedings of Mathematical Foundations of Computer Science 2003*, volume 2747 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2003. doi:10.1007/978-3-540-45138-9_15.

6 Bastien Cazaux and Eric Rivals. Reverse engineering of compact suffix trees and links: A novel algorithm. *Journal of Discrete Algorithms*, 28:9–22, 2014. doi:10.1016/j.jda.2014.07.002.

7 Julien Clément, Maxime Crochemore, and Giuseppina Rindone. Reverse engineering prefix tables. In *Proceedings of 26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009*, volume 3 of *Leibniz International Proceedings in Informatics*, pages 289–300, 2009. doi:10.4230/LIPIcs.STACS.2009.1825.

8 Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008. doi:10.1016/j.ipl.2007.10.006.

9 Maxime Crochemore, Costas S. Iliopoulos, Solon P. Pissis, and German Tischler. Cover array string reconstruction. In *Proceeding of 21st Annual Symposium on Combinatorial Pattern Matching, CPM 2010*, volume 6129 of *Lecture Notes in Computer Science*, pages 251–259. Springer, 2010. doi:10.1007/978-3-642-13509-5_23.

10 Jean-Pierre Duval, Thierry Lecroq, and Arnaud Lefebvre. Border array on bounded alphabet. *Journal of Automata, Languages and Combinatorics*, 10(1):51–60, 2005.

11 Jean-Pierre Duval, Thierry Lecroq, and Arnaud Lefebvre. Efficient validation and construction of border arrays and validation of string matching automata. *RAIRO Theoretical Informatics and Applications*, 43(2):281–297, 2009. doi:10.1051/ita:2008030.

12 Frantisěk Franěk, Shudi Gao, Weilin Lu, Patrick J. Ryan, William F. Smyth, Yu Sun, and Lu Yang. Verifying a border array in linear time. *Journal on Combinatorial Mathematics and Combinatorial Computing*, 42:223–236, 2002. doi:10.1.1.32.5012.

13 Pawel Gawrychowski, Artur Jez, and Lukasz Jez. Validating the Knuth-Morris-Pratt failure function, fast and online. *Theory of Computing Systems*, 54(2):337–372, 2014. doi:10.1007/s00224-013-9522-8.

14 Ira M. Gessel and Christophe Reutenauer. Counting permutations with given cycle structure and descent set. *Journal of Combinatorial Theory, Series A*, 64(2):189–215, 1993. doi:10.1016/0097-3165(93)90095-P.

15 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*. Cambridge University Press, Cambridge, United Kingdom, 1997. doi:10.1017/CBO9780511574931.

16 Jing He, Hongyu Liang, and Guang Yang. Reversing longest previous factor tables is hard. In *Proceedings of 12th International Symposium on Algorithms and Data Structures, WADS 2011*, volume 6844 of *Lecture Notes in Computer Science*, pages 488–499. Springer, 2011. doi:10.1007/978-3-642-22300-6_41.

17 Peter M. Higgins. Burrows-Wheeler transformations and de Bruijn words. *Theoretical Computer Science*, 457:128–136, 2012. doi:10.1016/j.tcs.2012.07.019.

18 Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Verifying and enumerating parameterized border arrays. *Theoretical Computer Science*, 412(50):6959–6981, 2011. doi:10.1016/j.tcs.2011.09.008.

19 Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Inferring strings from suffix trees and links on a binary alphabet. *Discrete Applied Mathematics*, 163:316–325, 2014. doi:10.1016/j.dam.2013.02.033.

20 Juha Kärkkäinen, Dominik Kempa, and Marcin Piątkowski. Tighter bounds for the sum of irreducible LCP values. *Theoretical Computer Science*, 656:265–278, 2015. doi:10.1016/j.tcs.2015.12.009.

21 Juha Kärkkäinen, Marcin Piatkowski, and Simon J. Puglisi. String inference from the LCP array. *CoRR*, abs/1606.04573, 2016. URL: http://arxiv.org/abs/1606.04573.

22 Gregory Kucherov, Lilla Tóthmérész, and Stéphane Vialette. On the combinatorics of suffix arrays. *Information Processing Letters*, 113(22-24):915–920, 2013. doi:10.1016/j.ipl.2013.09.009.

23 Udi Manber and Gene W. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

24 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows-Wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007. doi:10.1016/j.tcs.2007.07.014.

25 Yuto Nakashima, Takashi Okabe, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Inferring strings from Lyndon factorization. In *Proceedings of Mathematical Foundations of Computer Science 2014, Part II*, volume 8635 of *Lecture Notes in Computer Science*, pages 565–576. Springer, 2014. doi:10.1007/978-3-662-44465-8_48.

26 Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

27 Nicolas Philippe. Caractérisation et énumération des arbres compacts des suffixes. Master's thesis, Université de Rouen, 2007.

28 Klaus-Bernd Schürmann and Jens Stoye. Counting suffix arrays and strings. *Theoretical Computer Science*, 395(2-3):220–234, 2008. doi:10.1016/j.tcs.2008.01.011.

29 Imre Simon. Piecewise testable events. In *Proceedings of 2nd GI Conference on Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 214–222. Springer, 1975. doi:10.1007/3-540-07407-4_23.

30 Bill Smyth. *Computing Patterns in Strings*. Pearson Addison-Wesley, Essex, England, 2003.

31 Tatiana A. Starikovskaya and Hjalte Wedel Vildhøj. A suffix tree or not a suffix tree? *Journal of Discrete Algorithms*, 32:14–23, 2015. doi:10.1016/j.jda.2015.01.005.

32 Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.